INTELLIGENT CINEMATIC CAMERA CONTROL FOR REAL-TIME GRAPHICS APPLICATIONS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Ian Meeder

January 2020

COMMITTEE MEMBERSHIP

TITLE:	Intelligent Cinematic Camera Control for Real-Time Graphics Applications
AUTHOR:	Ian Meeder
DATE SUBMITTED:	January 2020
COMMITTEE CHAIR:	Zoë Wood, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Aaron Keen, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Franz Kurfess, Ph.D. Professor of Computer Science

ABSTRACT

Intelligent Cinematic Camera Control for Real-Time Graphics Applications

Ian Meeder

E-sports is currently estimated to be a billion dollar industry which is only growing in size from year to year [16]. However the cinematography of spectated games leaves much to be desired. In most cases, the spectator either gets to control their own freely-moving camera or they get to see the view that a specific player sees. This thesis presents a system for the generation of cinematically-pleasing views for spectating real-time graphics applications. A custom real-time engine has been built to demonstrate the effect of this system on several different game modes with varying visual cinematic constraints, such as the rule of thirds.

To create the cinematic views, we encode cinematic rules as cost functions that are fed into a non-linear least squares solver. These cost functions rely on the geometry of the scene, minimizing residuals based on the 3D positions and 2D reprojections of the geometry. The final cinematic view is found by altering camera position and angle until a local minimum is met.

The system was evaluated by comparing video output from a traditional rigidly constrained camera and the results of our algorithm's optimally solved views. User surveys are then used to qualitatively evaluate the system. The results of these surveys do not statistically find a preference between the cinematic views and the rigidly constrained views. In addition, we present performance and timing considerations for the system, reporting that the system can operate within modern expectations of latency when enough constraints are placed on the non-linear least squares solver.

ACKNOWLEDGMENTS

Thanks to:

- Dr. Zoë Wood for constantly encouraging me to finish this thesis despite my best efforts. Without her patience and kind words it wouldn't have been completed.
- My parents for instilling in me a need to succeed academically from an early age.
- My uncle for helping my financially and making sure I was able to graduate without a mountain of debt.
- All my friends whose sarcastic derision kept me humble.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

Chapter 1

INTRODUCTION

Art can be broken down into two key aspects: the content, and how the content is presented to the audience. No matter the format, the plot of a story is unchanging - Odysseus' journey did not change in between the original oration and the written prose; at its essence it is the same. However the way in which we consume the Iliad, or any art, is directly related to how it is presented. A spoken poem may include vocal intonations that are missing from its written form, and upon reading the poem we can see the beauty behind the syntactic construction.

In film, the way in which the story is presented to the audience is called the cinematography. It is how the content of the film fits within the frame, the duration of the shots, the camera angles, and the transitions. While filming, the director and cinematographer will deliberately place and move the camera, each decision quite intentional. In the way that an author must follow the strict rules of the language in which they write, a director, too, will follow rules of cinematography. And like the author, the director will also know when to break those rules.

These rules don't just apply to film, but to video games as well. Usually, the player controls the virtual camera within a game. However it is also common for this control to be ripped away for brief periods when the developer wants to force the player to see a specific view. This view can be within the game engine or a cinematic cutscene that was created before-hand and played back to the player like a movie. And in nearly every one of these situations, the developer will present this view within the basic rules of cinematography.

Streaming sites like Twitch are blowing up in popularity and e-sports is quickly growing into a billion dollar industry [16]. These platforms however are primarily focused on player perspective or at the very least in the case of e-sports, a moderator driven one. A unique opportunity presents itself — can we improve the cinematic qualities of spectating video games?

1.1 AestheticCam

This paper presents an intelligent virtual camera system designed to produce automated cinematic views, called AestheticCam. Given context of the virtual environment in conjunction with game-specific logic, the camera produces an intelligently-chosen view for the given situation.

Picking views is determined by constraints and heuristics based on universally-accepted rules of cinematography and situational rules of the game. Generally speaking, these rules apply to:

- framing - where to place an object of interest within the frame
- spatial placement - placement of the camera within the virtual environment
- timing - how long to perform a particular shot

The camera system is embedded within a larger C++ framework and OpenGL framework. This framework manages the overall virtual environment to update and draw to the screen. While integrated into the system, AestheticCam operates independent from gameplay, producing views designed with cinematic integrity in mind without interfering with the player.

Chapter 2

BACKGROUND

2.1 Computer Graphics

The application of Computer Science towards generating images is known as Computer Graphics. This discipline allows the video game and movie industries to create fantastical worlds with realistic fidelity. While a lot of modern graphics standards are reliant on the abilities of artists, the foundation of these effects are still heavily reliant on mathematics. To represent objects in a virtual world, computer graphics relies on a series of interconnected vertices called meshes. The vertices then undergo a series of linear algebraic transformations which re-position and animate the mesh. Virtual cameras exist within such an ecosystem and the design of our cinematic camera system requires a strong understanding of the fundamentals behind computer graphics applications.

2.1.1 Triangle Meshes

While not the sole method for representing objects in a three-dimensional space, triangle meshes are certainly the most prominent. Hearkening back to the basics of Euclidean geometry, any geometric object can be represented by a set of faces. Faces are composed of interconnected vertices and those connections are called edges. Objects in the context of computer graphics are represented the exact same way, and these are called polygonal meshes or simply meshes. A triangle mesh is a special kind of mesh in which every face is a triangle - which has a special significance for computer graphics. In order to represent the idea of a mesh computationally, the

vertices of a mesh are stored as x, y, and z coordinates within a three-dimensional Cartesian coordinate system.

Apart from positional data, a mesh often contains additional three-dimensional and surface-localized two-dimensional data to be used for rendering. For example, vertex normals are a three-dimensional representation of the surface normal at a vertex and are used to compute surface smoothness. As for two-dimensional vertex data, texture coordinates are stored per-vertex to map vertices into a normalized two-dimensional coordinate system. This mapping is used to sample a variety of images, all of which are used to improve the quality of the final render.

2.1.2 Linear Algebra

Each vertex of a triangle mesh is defined within its own local coordinate frame. In order to allow an object to move around and rotate within a larger three-dimensional world, basic transformations need to be applied to each individual vertex of a triangle mesh; two of the most basic transformations being translating and rotating. The translation of a mesh is stored as the displacement from its local coordinate frame into the world frame, which we'll refer to as T. A myriad of representations exist for the rotation of an object in three-dimensional space, such as quaternions or rodrigues. However a common one (and easy to wrap one's head around) is Euler angles. Euler angles, ψ, θ, and ϕ represent the rotation of an object about its local coordinate frame's x, y, and z axes respectively. Both translation and rotation are used to construct a single model to world transformation matrix, which for our purposes we

will refer to as M. Because matrix multiplication is not commutative, it is important to understand that Equation 2.1 first applies rotation about Z, then X, and finally Y.

$$M = \begin{bmatrix} cos\theta cos\phi + sin\theta sin\psi sin\phi & sin\theta sin\psi cos\phi - cos\theta sin\phi & sin\theta cos\psi & T_x \\ cos\psi sin\phi & cos\psi cos\phi & -sin\psi & T_y \\ cos\theta sin\psi sin\phi - sin\theta cos\phi & cos\theta sin\psi cos\phi + sin\theta sin\phi & cos\theta cos\psi & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

2.1.2.1 Virtual Camera

Part of transforming a vertex from its local object coordinate frame into clip space is the transform that places that vertex in the coordinate frame of a virtual camera. A virtual camera represents a six degree-of-freedom rigid body in three-dimensional space. In computer graphics, the camera mainly represents an affine transform that converts objects in world space to view space. Similar to how Equation 2.1 transforms vertices from a model coordinate frame into world coordinate frame, the inverse of a virutal camera's M transform (Equation 2.2) gives us a transform from world into the camera or view space. Once in view space, objects are reprojected into the view frustum.

$$V = M_{camera}^{-1} \tag{2.2}$$

2.1.2.2 View Frustum

The view frustum is a volumetric representation of what the camera is looking at. If you take the camera's position and create a square pyramid-shaped volume pointing in the direction that the camera is looking, you have the basis for view frustum. Because of computer limitations, a limit is placed at how far the frustum will extend

from the camera, making up a far plane; objects further from the camera than the far plane will not be rendered. The image plane is the surface objects are projected onto. This is inserted very close to the position of the camera and is also called the near plane Using the information of the distance of the near and far planes from the camera, and the angle of the sides of the pyramid — called the field of view — any vertex positioned within the frustum can be mapped to the -1 to 1 limitation of the view plane coordinates. For a more succinct picture, look at Figure 2.1.

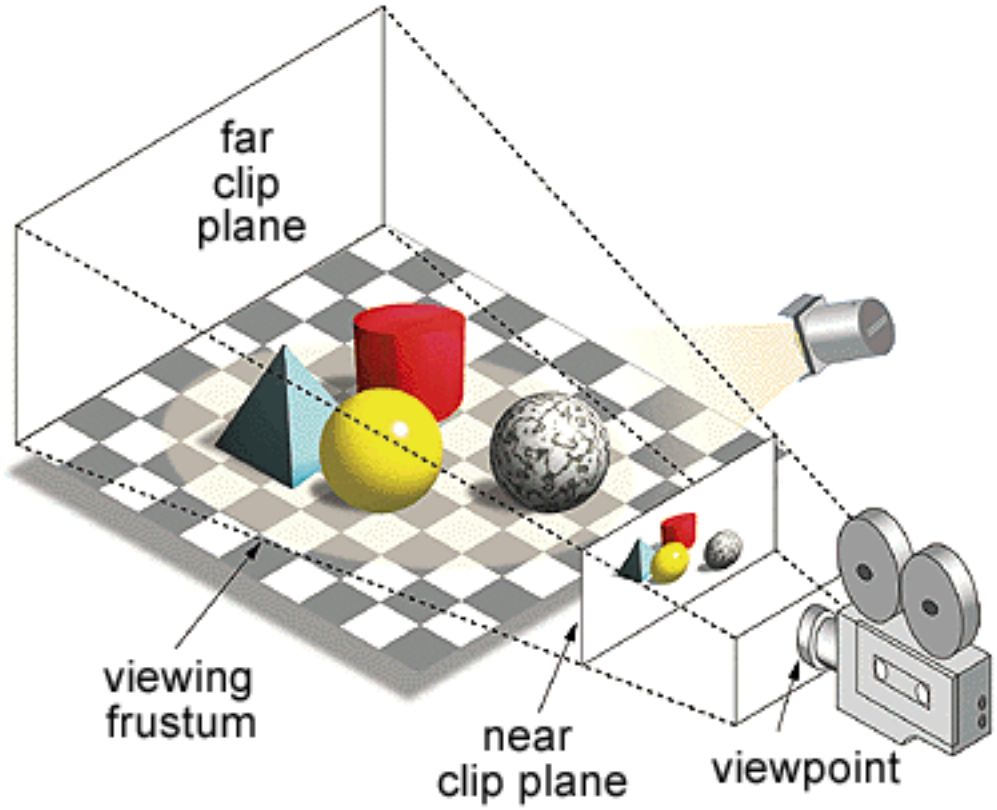


Figure 2.1: A representation of how the view frustum contains other meshes in 3D space. [14]

$$P = \begin{bmatrix} \frac{1}{tan(fov/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{tan(fov/2)} * aspect\_ratio & 0 & 0 \\ 0 & 0 & \frac{near + far}{near - far} & \frac{2 * near * far}{near - far} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{2.3}$$

By breaking the view frustum into a field of view, near plane distance, far plane distance, and aspect ratio, a projection matrix, P can be constructed which will project points in view space into clip space (see Equation 2.3).

2.1.3 Graphics Libraries

In order to display our three-dimensional meshes as rendered images, we make use of low-level graphics libraries. These libraries enable developers to write programs called shaders that run on graphics cards to manipulate primitive geometric objects, such as triangles, and render them to the screen. This thesis makes use of OpenGL, but the principles outlined here are extensible to all other libraries like DirectX, Metal, or Vulkan.

2.1.3.1 OpenGL Pipeline

In order to render images to the screen, OpenGL gives control of the GPU via the OpenGL Shading Language (or GLSL). Shaders written in GLSL are converted into executable programs that are run at specific times during OpenGL's rendering pipeline (see Figure 2.2). The most ubiquitous steps of this pipeline are the vertex shader and the fragment shader.

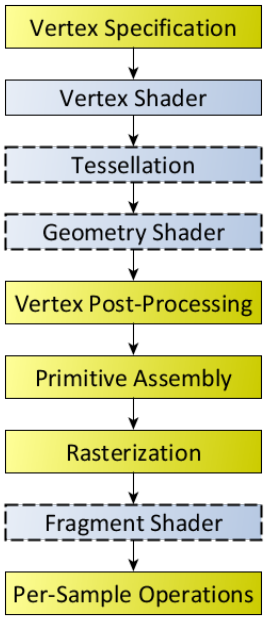


Figure 2.2: The OpenGL rendering pipeline. Blue boxes represent programmable stages. [24]

2.1.3.2 Vertex Shader

The job of a vertex shader is to take incoming vertex data and perform relevant computations that are then passed further down the OpenGL rendering pipeline. Triangle meshes are uploaded to the GPU and arranged such that data is clumped together per-vertex. This entails such data as the vertex's Cartesian position or its texture coordinate as was discussed in Section 2.1.1. The primary goal of a vertex shader is to convert each vertex's coordinate system into one that OpenGL can natively digest. This coordinate system is known as clip-space — a cube in a 3D homogeneous coordinate space ranging from (-1, -1, -1) to (1, 1, 1). Using transformation matrices outlined in Equations 2.1, 2.2, and 2.3, we are able to transform these vertices — and more specifically the triangles they represent – into clip-space which are rasterized by OpenGL and sent along to the fragment shader.

2.1.3.3 Fragment Shader

The rasterization process takes the triangles that were output from the vertex shader and converts them into pixels (or sub-pixel fragments) to perform further computations on. The output of the fragment shader is a value for the color of a particular pixel. Therefore the computations being performed in the fragment shader are generally for the purpose of determining how a pixel should be lit in the final rendered scene. However, more advanced techniques will use the fragment shader to produce additional information, encoding it within the final rendered image.

2.2 Non-Linear Least Squares

Our cinematic camera system needs some way to determine its behavior. For any given configuration of the elements in the scene, the camera should try and find the most optimal position and rotation. We accomplish this by modeling the desired behaviors as a non-linear least squares optimization problem.

Non-linear least squares is defined by the following equation:

$$F(x) = \frac{1}{2}\sum_{i=1}^{m} r_i^2(x) \tag{2.4}$$

Where r_i represents a smooth function that maps $\mathbb{R}^n \to \mathbb{R}$ where $m \geq n$ [22]. Each function r_i is also known as a *residual* of the overall optimization problem. By minimizing $F(x)$, non-linear least squares optimizers find the set of valid parameters that create an optimal model to fit the given data, such as in Figure 2.3.

To find a minimum, non-linear least squares optimizers take an iterative approach [18]. Starting with a set of initialized input parameters, each iteration of the optimizer

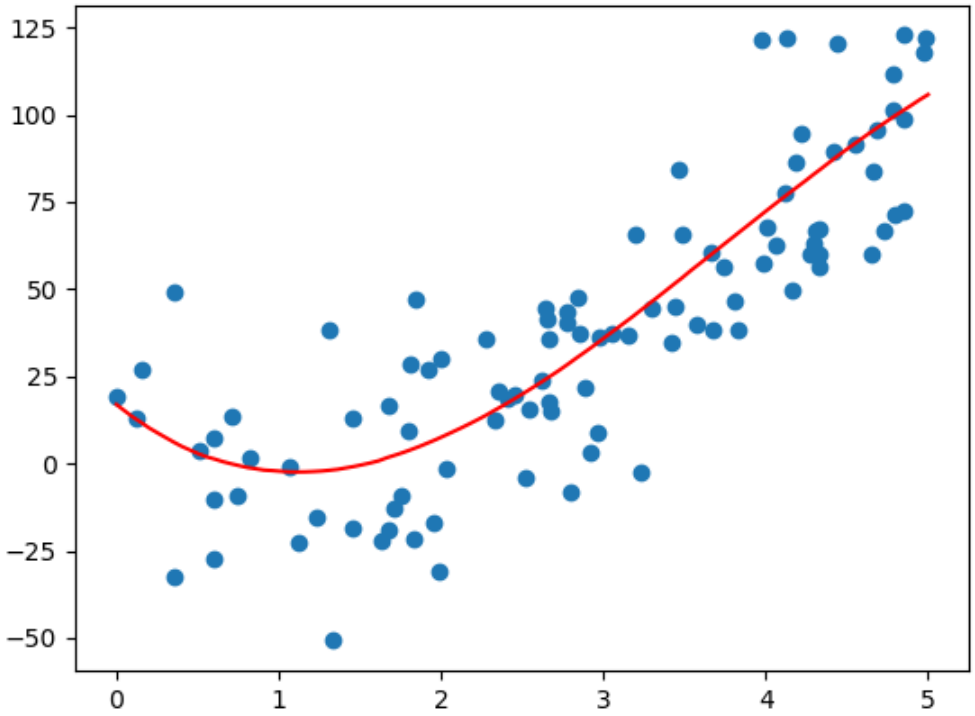


Figure 2.3: A typical use case for non-linear least squares is to find best-fitting lines for polynomial regressions. Above is a sample cubic regression used to fit a generated data set.

will produce a new set of inputs with the hope that the new inputs cause $F(x)$ to converge on some local minimum.

2.2.1 Line Search

The key to an efficient optimizer is how efficient the descent method is — how quickly it can reach a local minimum with confidence. One common optimization strategy is known as line search. In line search, each iteration of the optimizer for some step k is computed as:

$$x_k = x_{k-1} + \alpha h \tag{2.5}$$

Where h represents a vector in the descent direction and α representing a scalar distance.

A common way to choose h is to use the first derivative of $F(x)$ and use the gradient of the function as h. By computing the first partial derivative of each r_i in the input space, the Jacobian Matrix, $J(x)$, of $F(x)$ is constructed. The gradient of $F(x)$ at some x_k, can then be computed as:

$$h_k = -J(x_k) \tag{2.6}$$

This approach is called steepest descent or gradient descent. [18].

Once the optimal h value has been found, getting an appropriate α is next on the list. Finding the local optimal α along h can be an expensive operation as it requires evaluating the cost function at many different points. To curtail unnecessary iterations, a variety of heuristics are used to decide when "good enough" is good enough.

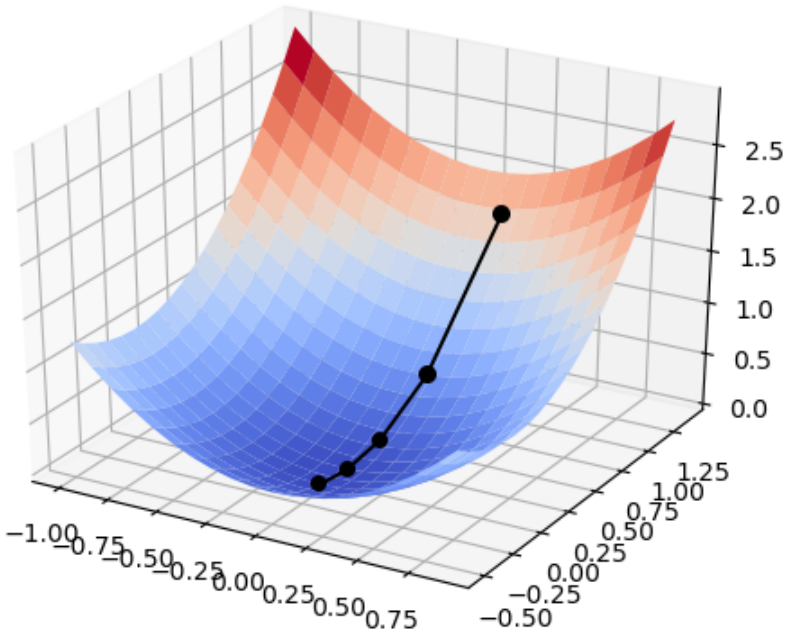


Figure 2.4: A rudimentary depiction of gradient descent for $f(x, y) = x^2+y^2$. Direction is determined by the gradient of the function and the size of the step is determined by a series of heuristics including the magnitude of the directional derivative of the function along the gradient.

One set of heuristics is called the Wolfe Conditions [22]. The first Wolfe Condition is called the sufficient decrease condition. The sufficient decrease condition stipulates that α must satisfy the following inequality:

$$f(x_k + \alpha h_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T h_k \tag{2.7}$$

or in other words, the destination of the current iteration must be less than the previous iteration proportional to α and the directional derivative [22]. Alone, this condition can be trivial to achieve, so an additional Wolfe Condition is used called the curvature condition, which is written as:

$$\nabla f(x_k + a_k p_k)^T h_k \geq c_2 \nabla f_k^T h_k \tag{2.8}$$

This condition states that the slope at α should be relatively flat or positive before deciding on a good α value for the current iteration. Intuitively this makes sense because if the slope at α was very negative, a better local minimum could be achieved by making α larger, continuing in the direction of h.

Using these heuristics, α values are selected with an iterative approach. Each set of h and α gets the optimizer closer to finding a local minimum and eventually settles on a set of optimal input parameters. Although seemingly initially paradoxical, it is this set of input parameters that comprise the final output of the optimizer.

2.3 Computational Aesthetics

While we may have an innate idea of what is aesthetically pleasing, the mathematically-inclined have been trying to quantify art for decades. Taking a more mathematical approach to aesthetics and cinematography helps inform the final design of our camera system.

In 1928, George D. Birkhoff coined "aesthetic measure" as the ratio between order and complexity of a work [23]. From this definition, researchers birthed the field of "informational aesthetics" in an attempt to quantify what order and complexity mean in the context of visual art [23]. Now that computers rule the modern era, computer vision experts have taken to the problem and coined their own sub-field called computational aesthetics.

In 2005, Neumann et al. defined computational aesthetics as "the research of computation methods that ... make applicable aesthetic decisions in a similar fashion as humans can" [21]. Hearkening back to its origins, computational aesthetics is still most commonly used to analyze the aesthetic quality of in image, rather than their creation. Datta et al. created an extensive computational model to predict the aesthetic quality of an image [8]. This model was based on exposure, colorfulness, image composition, and depth of field to name a few. They discovered a strong correlation between these low-level visual indicators and perceived aesthetic merit [8].

2.4 Cinematography

Most of the work done by computer vision scientists for computational aesthetics has been applied to static images. Many of the same principles transfer to video and now fall under the umbrella of cinematography. Like all art, cinematography is subjective, but understanding where and why directors and cinematographers make the decisions they make has been extensively studied and written about.

While the intricacies of cinematography are vast and minute, Joseph Mascelli boils the basics down into five fundamental points: Camera Angles, Continuity, Cutting, Close-Ups, and Composition [20]. A subset of both camera angles and composition,

a metric that rears its head time and time again in both cinematography and computational aesthetics is the rule of thirds.

2.4.1 Rule of Thirds

The rule of thirds starts by dividing a frame both vertically and horizontally three times, creating a tic-tac-toe grid [5]. It is at the intersection of these grid lines that the subject of a shot will be placed (see Figure 2.5). This method is so ubiquitous that it was even an explicit metric defined by the aesthetic model from Detta et al. in Section 2.3 [8].

Figure 2.5: Rule of thirds is used in both film and photography. The ratios between empty space and the subject caused by adhering to this rule is a common but effective crutch [19].

Chapter 3

RELATED WORKS

3.1 Camera Planning

While many factors can play into real-life cinematography such as lighting and staging, virtual systems generally have less control when it comes to on-the-fly cinematography. The most control these systems have is over camera placement and orientation. The classification of camera planning systems has been defined as such:

- algebraic systems represent the problem in vector algebra and directly compute a solution;
- optimization and constraint-based systems model the properties as constraints and objective functions and rely on a broad range of solving processes that differ in their properties (e.g. completeness, incompleteness, and softness). [7]

3.1.1 Algebraic Systems

Systems that adhere to the algebraic classification are those who have direct solutions to a mathematical problem. As such, these systems tend to be very rigid. The first example of such a system can be attributed to Blinn in 1988 [3]. Blinn's goal was to take interesting pictures of space with a spaceship in the foreground and a planet or some other celestial body in the background. As a result, the images were very much the same and were meant to come up with a very specific image according to the vector algebra involved.

Attempts to create less rigid algebraic systems have been made. Christianson et al. created such as system that abstracted rules of cinematography into what they

called idioms [6]. Each idiom would be applied if the state of the world coincided with the prerequisites for that idiom to occur. However, this system was still rather rigid.

3.1.2 Optimization and Constraint-based

Constraint-based systems can be viewed as an optimization problem for dynamic camera planning. Theses systems make distinctions between firm constraints that a camera must follow as well as properties that a particular shot has that are to be minimized/maximized. Generally these systems will "try out" a series of shots, and choosing the most optimized according to the constraints.

Although complete solutions have been proposed (in which the entire solution space is sketched out), incomplete systems allow for more flexibility and allow for more constraints

Drucker et al. implemented their system CINEMA in 1992 which was an early attempt to create a soft constraint-based camera movement system [10]. Based on these efforts, authors Drucker and Zeltzer created Virtual Museum where a virtual camera would traverse a virtual museum, giving a "tour" of the museum [11]. This improvement on the system allowed for a more goal-oriented approach to movement planning. Continuing this work, Drucker and Zeltzer created CamDroid, which encapsulated desired camera-related tasks into modules that allowed for more extensible behavior [12].

In 1998, Bares et al. came up with constraint based camera behavior that focused on intelligent constraint-based cinematography [2]. Their system, ConstraintCam, allowed for real-time modifications of an interactive fiction environment. ConstraintCam was able to follow various cinematic rules such as view angle and shot distance

from user-specified subjects. If a shot that was not good enough was found, then a default overhead shot was taken.

3.2 Probabilistic Roadmaps

Probabilistic roadmaps (PRM) is a robotics algorithm in which movement paths are generated as random weighted undirected graphs [17]. The robot's movement is then chosen from this graph using a series of heuristics and the weights of the generated graph. In Davis' thesis titled "Probabilistic Roadmaps for Virtual Camera Pathing with Cinematographic Principles", probabilistic roadmaps in conjunction with computer vision techniques were used to select cinematic camera paths within a virtual environment [9]. Using blob detection techniques, generated nodes of the PRM are given weights based on the rule of thirds. After the generation of the PRM, an optimal camera path is generated by selecting the best nodes in the graph based on a mix of heuristics and the computer-vision evaluated rule of thirds metric. Since blob detection requires an image to evaluate, each node in the PRM requires a render pass which adds overhead and does not allow a PRM approach to be run in real-time; paths from Davis' evaluation took anywhere 6.4 to 38.5 seconds to generate.

3.3 Human-Driven

Due to the very artistic and subjective nature of cinematography, automated cinematic camera systems are often designed with a huge degree of human input. To bridge the gap is Cambot, a lightweight system developed by Elson and Reidl to mimic real filmmaking processes for virtual environments [13]. Cambot is a scriptable cinematic camera system with built-in knowledge of a set of standard camera movements, such as wide shots and shot/reverse-shots for character dialog. Cambot's

knowledge is specific and its behavior is limited by its set of known shots as well as the ability of the "director" to properly script Cambot.

3.4 Unmanned Aerial Vehicles

Unmanned aerial vehicles (UAVs), also known colloquially as drones, provide directors with unique angles that are not constrained in the same ways as standard filming methods, such as steadicam, cranes, or gimbals [15]. In many ways, UAVs are as free as virtual camera systems, allowing many of the same approaches for automated cinematography to apply to both. In 2017, Galvene et al. designed a system that relies on external tracking data and user input to generate cinematic paths for UAVs [15]. This nameless system relied on many of the same principles that the constraint-based systems in Section 3.1.2 established. By treating the real world as a virtual one via use of tracking systems and virtual placeholders, camera control systems can be utilized for both virtual and real world applications.

Chapter 4

SYSTEM DESIGN

The intent of the AestheticCam system is to interface with an existing game engine with as little engine-specific elements as possible. In the spirit of this design, the work presented in this thesis can be broken into two fairly disjoint parts — the game engine and the AestheticCam solver.

4.1 Game Engine

Game engines created by companies can vary wildly depending on the game type being developed. However the general pieces are always present, such as the rendering system. We developed our own game engine in order to focus on the cinematic camera system and avoid the complexities of integrating with an existing game engine. The game engine developed for this thesis relies on a home-brewed entity-component (ECS) architecture with an OpenGL-driven rendering system and physics system — or more specifically collision detection system — handled by the Bullet Physics library.

4.1.1 Overview

Game engines are driven by a never-ending loop called the game loop. Each iteration of the game loop is broken into two phases — Update and Draw. During the update phase, game entities are spawned, destroyed, and moved according to game-specific logic and the physics system. Once all updating is finished, the scene is rendered to the screen in the Draw phase. Then the process repeats. Modern games accomplish

this loop at upwards of 60 times per second which creates the illusion of a smooth video.

Algorithm 4.1: Game Loop

```
prev ← Now();
while true do
    // Compute the time elapsed during the previous game loop
    now ← Now();
    time_elapsed ← now - prev;
    prev ← now;
    // Call all entities in the scene to update their positions
    Update(scene, time_elapsed);
    // Call any async scheduled tasks on the render thread.
    PollDispatchedTasks()
    // Determine which entities are colliding
    collisions = ComputeCollisions(scene);
    // Resolve the collisions (e.g.  push a player out of a wall)
    ResolveCollisions(collisions);
    // Render the entities to the screen
    Draw(scene);
end
```

4.1.2 Entity Component System

Game development, like modern application development, takes a very object-oriented programming (OOP) approach. Entities in the scene require a set of standard attributes - position, rotation, scale - so it makes sense for all entities to inherit from some generic parent class, Entity. However these entities will then require specific

game logic to create a compelling and unique game. An OOP solution would be to implement the specific game logic in a child of the Entity class. This solution can get messy or repetitive as certain game logic is universal across many entities while specific logic may only show up once. Instead, a more extensible solution is to encompass game logic in a separate class that can be called from the Entity class (i.e. the strategy pattern). This class, called Component, defines a function that can be overridden which will be called every game loop. Each Entity contains n Components, allowing for individual units of game logic to added and removed from entities in bite-sized pieces.

For example, a class called PlayerController may inherit from Component and implement a function called `Update(float time_elapsed)` which moves the owning Entity when specific keys or buttons are pressed by the player. PlayerController can then be added to any Entity that should be controlled by the player. Components are not limited to just logic, but can also extend the attributes of an Entity type, such as providing velocity for an entity with PhysicsComponent or adding a MeshComponent to be drawn by the rendering system. See Figure 4.1 for an example of the class diagram.

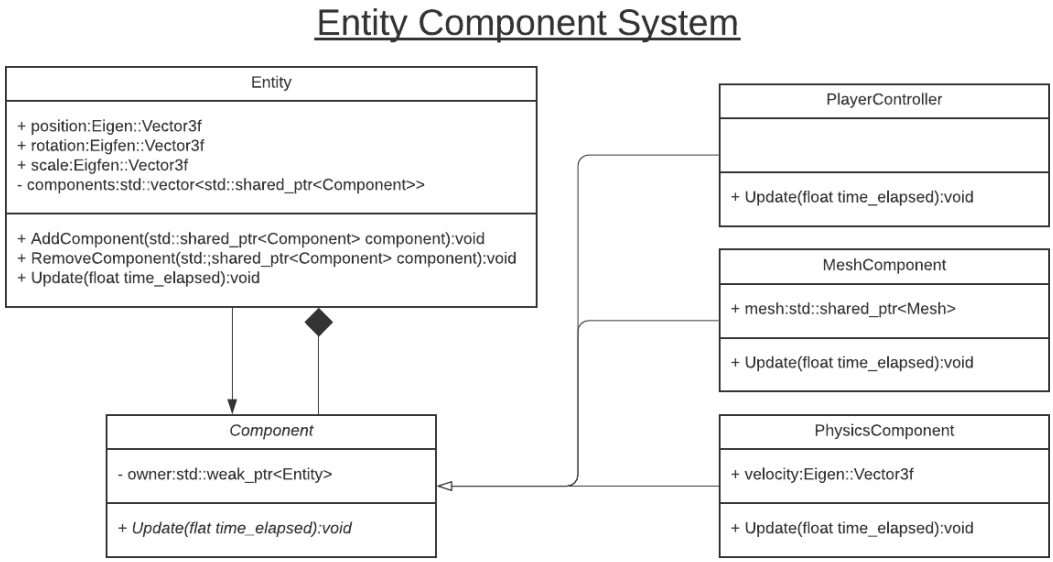


Figure 4.1: Example class diagram for the entity-component system.

4.1.3 Physics

Depending on the type of game, a physics system is often required. While a strategy or puzzle game might not require physics, platformers and first person shooters rely on physics systems to create convincing experiences. For the purposes of this thesis, realistic physics wasn't strictly necessary. Instead the only thing the physics system really needed was the ability to handle objects colliding. Collision is broken into two steps — collision detection and collision resolution.

4.1.3.1 Collision Detection

Collision detection is handled by the open-sourced Bullet Physics library. Every entity that requires collision is given a ColliderComponent which determines the type and shape of collision object that Bullet can consume. Supported shapes include capsules, spheres, boxes, and convex hulls. Every iteration of the game loop constructs a

new snapshot of where the collision objects are, which is then passed into Bullet's discrete collision detector. For every two objects that collide, Bullet returns a struct representing the collision.

4.1.3.2 Collision Resolution

After Bullet has computed all collisions, the system decides on what to do with the collision information. Entities can contain CollisionResponse components which are called when the owning entity has collided with another object. This behavior is used for game logic responses that don't follow the standard physics simulation, such as a player colliding with a projectile. Outside of CollisionResponse components, there is a standard physics response which will ensure that moving entities are forced to remain outside static entities such as walls.

4.1.4 Rendering

Once all physics collisions have been resolved, we are then ready to render all drawable game entities to the screen. This responsibility falls under the rendering system, which relies on OpenGL shaders as talked about in Section 2.1.3. By following advanced graphics techniques, the rendering system creates a final visual output that is expected of a modern video game.

4.1.4.1 Blinn-Phong

The rendering system relies on a set of deferred GLSL shaders to compute mesh lighting and shadows. To accurately depict rendered surfaces, graphics shaders will rely on a bidirectional reflectance distribution function (BRDF) that defines how the light of a scene will reflect off a surface. The rendering system used in this thesis relies on a relatively mathematically simplistic model called Blinn-Phong [4]. This equation

is outlined in Algorithm 4.2. Roughly, the Blinn-Phong BRDF computes the lighting for any given point on a surface by taking the average of the light vector L, and the view vector V to approximate the angle of specular reflectance. In conjunction with material properties for color, K_a, K_d, and K_s, which represent the base albedo for ambient, diffuse, and specular illumination, respectively, and α, which represents the measuere of shininess of the material, the Blinn-Phong model can represent a wide variety of different objects.

Algorithm 4.2: Blinn-Phong Model for a point on a surface

Result: The total illumination for the given point

H $\leftarrow \frac{L+V}{2}$;

Diffuse $\leftarrow$ max(0, L $\cdot$ N) * K_d;

Specular $\leftarrow$ pow(max(0, H $\cdot$ N), α) * K_s;

I $\leftarrow K_a$ + Diffuse + Specular;

return I;

4.1.4.2 Deferred Rendering

Besides using a Blinn-Phong BRDF, additional real-time rendering techniques were used to approximate a modern game engine. Simple rendering pipelines will pull off the desired result in a single render pass; this is called a forward renderer. However modern graphics standards often require multiple rendering passes to accomplish the desired effect, called a deferred renderer. In addition to enabling more advanced techniques, such as screen space ambient occlusion, deferred rendering can also be more efficient rendering scenes with multiple lights and hundreds of different objects. Simple computations are front-loaded and stored in buffers which are used as inputs to subsequent render passes with the final render pass performing the final (and more expensive) BRDF lighting computations.

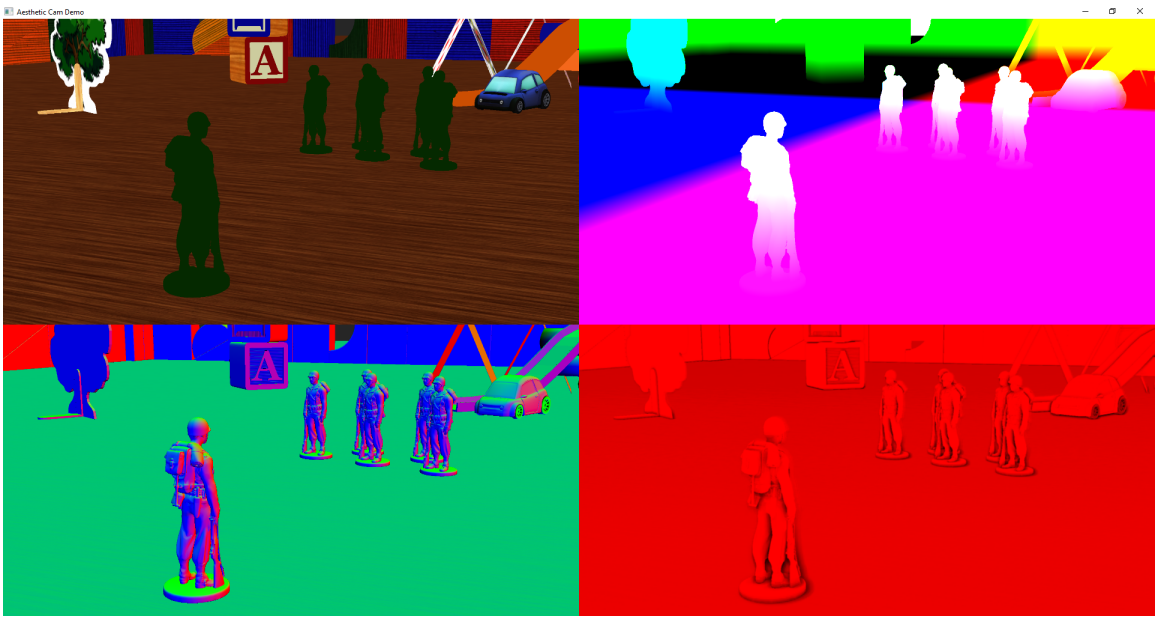


Figure 4.2: The result from initial deferred rendering passes. Surface albedo (top-left), position (top-right) and surface normal (bottom-left) are rendered in a single pass and used to compute screen space ambient occlusion (bottom-right) before the final lighting computation.

4.1.5 Multi-threading

In a typical game engine, all rendering is accomplished by a single thread. However as the quality of the render goes up, so does the time it takes to render. For a modern game to achieve a 30 frames per second frame rate, the entire game loop must take less than 33 milliseconds to finish. As games become more and more complicated, a 33 millisecond window isn't enough time to both render and carry out more complex computations like physics. The solution is to move the costlier computations to a separate thread from the rendering and reserve the main thread purely for rendering. Our system accomplishes this by utilizing Boost's asynchronous I/O library to create a queue-like threadpool implementation. Tasks are dispatched to the threadpools and executed in parallel. An additional single-threaded queue is reserved for the main thread and executed in series with the rest of the game loop to ensure thread safety.

4.1.6 Shooter Game

To fully test the cinematic camera system, we built a first-person shooting game using the game engine. The game involved a player controlled character being chased by a team of enemy characters within an enclosed environment. This environment contains obstacles that the player can run around and jump on top of while the enemy characters chase the player character. The player has the option of simply running away, or to lob projectiles at the enemies. Enemies hit with a projectile are defeated and vanish from the game (see Figure 4.3).

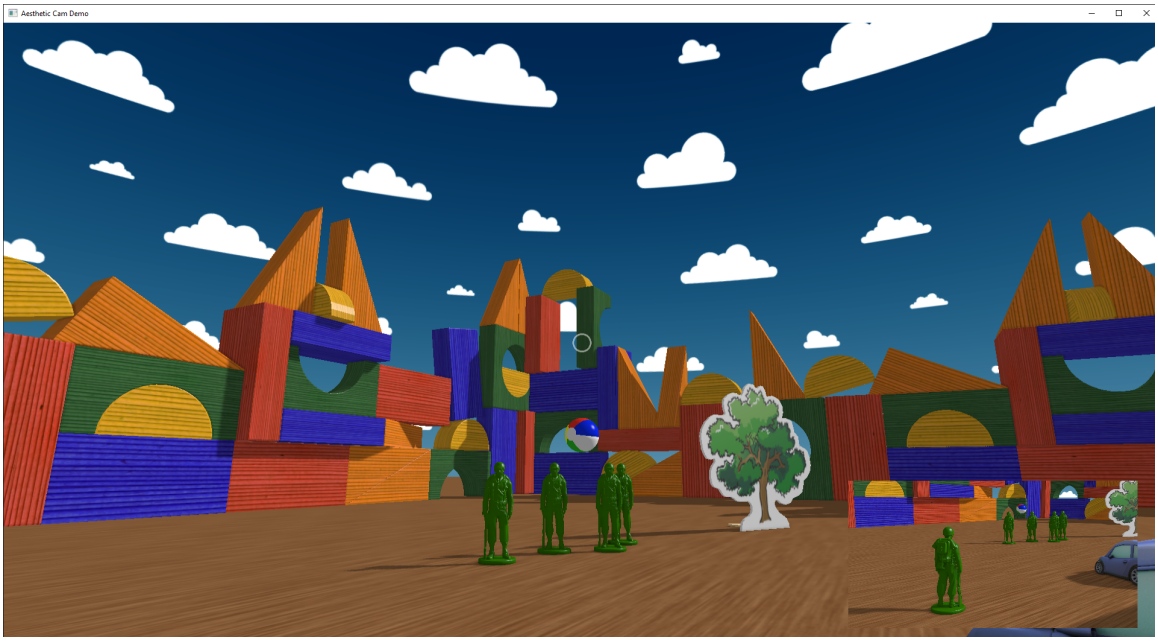


Figure 4.3: Screenshot from the shooter game. Here the player has fired a projectile (a beach ball) and taken out one of the five enemy units.

4.2 AestheticCam

In an attempt to continue the work of the optimization and constraint-based systems of Section 3.1.2, this thesis proposes a non-linear least squares-based camera system called AestheticCam. AestheticCam is meant to operate as independent of the under-

lying engine as possible while providing cinematic camera angles in real-time. Despite this design philosophy, AestheticCam's implementation still caters to the entity component system (see Section 4.1.2) dictated by the game engine. AestheticCam was then specifically designed to create cinematic views for the game described in Section 4.1.6.

4.2.1 Ceres

AestheticCam relies on Ceres Solver to perform all its non-linear least squares optimizations [1]. As with most non-linear least squares optimizers, the underlying system takes an iterative approach to optimization, utilizing gradient descent techniques as seen in Section 2.2.1.

Ceres takes a modular approach to creating an optimizable solver, allowing developers to create individual cost functions that can map any number of parameters to any number of residual values. As described in Section 2.2, a cost function is described as a function $f : \mathbb{R}^n \to \mathbb{R}$, however creating an individual cost function for each residual would be impractical so Ceres allows us to create cost functions that map to m residual values, i.e. $f : \mathbb{R}^n \to \mathbb{R}^m$.

In game engines, entities are typically given free range of motion within the world. This equates to six degrees of freedom, however camera roll can often get unwieldy or disorienting in virtual systems. As such, AestheticCam limits camera parameters to only 5 degrees of freedom, eliminating roll. In terms of Ceres cost functions, the input size is 5 which can map to m residual values, i.e. $f : \mathbb{R}^5 \to \mathbb{R}^m$.

4.2.2 Auto Differentiation

As outlined in Section 2.2, the gradient descent method employed by Ceres requires both residual values as well as the Jacobian matrix for that residual. For a cost function $f : \mathbb{R}^5 \rightarrow \mathbb{R}^m$ where $m = 2$, the following matrix would need to be computed:

$$\begin{bmatrix} \frac{\partial_{m_1}}{\partial_x} & \frac{\partial_{m_1}}{\partial_y} & \frac{\partial_{m_1}}{\partial_z} & \frac{\partial_{m_1}}{\partial_\phi} & \frac{\partial_{m_1}}{\partial_\theta} \\ \frac{\partial_{m_2}}{\partial_x} & \frac{\partial_{m_2}}{\partial_y} & \frac{\partial_{m_2}}{\partial_z} & \frac{\partial_{m_2}}{\partial_\phi} & \frac{\partial_{m_2}}{\partial_\theta} \end{bmatrix} \tag{4.1}$$

However, analytically computing the partial derivatives which compose the Jacobian matrix can be cumbersome. With large and complex residual functions, computing the Jacobian matrix gets exceedingly onerous when attempting to iterate; a simple change to the residual could mean completely re-computing the Jacobian matrix. Fortunately, Ceres makes use of a technique called auto differentiation, permitting us to completely avoid computing the Jacobian matrix.

4.2.2.1 Dual Numbers

In the same way that a complex number consists of some real component and an imaginary component, Ceres uses the idea of a *dual number* which consists of a real component, a, and an infinitesimal component, $v\epsilon$, where $\epsilon^2 = 0$ [1]. Expressed similar to a complex number, we get $a + v\epsilon$.

4.2.2.2 Taylor Series

Taylor series expansion is a method for expressing a function near some point x. For some value a close to x, the Taylor expansion of a function f is expressed as:

$$f(a) = f(x) + f'(x)(a-x) + \frac{f''(x)}{2!}(a-x)^2 + \frac{f'''(x)}{3!}(a-x)^3... \tag{4.2}$$

If we take a cost function f, we can express $f(x+\epsilon)$ as a Taylor series near x:

$$f(x+\epsilon) = f(x) + f'(x)(x+\epsilon-x) + \frac{f''(x)}{2!}(x+\epsilon-x)^2 + \frac{f'''(x)}{3!}(x+\epsilon-x)^3... \tag{4.3}$$

And because of our definition of $\epsilon^2 = 0$, Equation 4.3 conveniently condenses down to:

$$f(x+\epsilon) = f(x) + f'(x)\epsilon \tag{4.4}$$

4.2.2.3 Jets

Expanding upon the idea of dual numbers, Ceres makes use of *jets*, a dual number with an n-dimensional infinitesimal component, expressed as $a + \sum_{j=1}^{n} v_j\epsilon_j$ where any two $\epsilon_i\epsilon_j = 0$. The Taylor expansion of this multi-dimensional Jet similarly gives us:

$$f(x+\epsilon) = f(x) + f'(x)\sum_{j=1}^{n} v_j\epsilon_j \tag{4.5}$$

Applying this same logic to our $f : \mathbb{R}^n \to \mathbb{R}^m$ Ceres cost functions gives us:

$$f((x + \sum_{j=1}^{n} v_j\epsilon_j)_1, ..., (x + \sum_{j=1}^{n} v_j\epsilon_j)_n) = f(x_1, ..., x_n) + \sum_{j=1}^{n} f'_j(x_1, ..., x_n)\epsilon_j \tag{4.6}$$

Once the equation makes it to this final iteration, we find that the coefficient of each ϵ_j gives us the partial derivatives we need in order to construct the Jacobian matrix of our cost functions. Through clever overriding of basic mathematical operators in C++, jets can keep track of the ϵ coefficients without having to change how we write our cost functions [1].

4.2.3 Extending the Existing System

Despite running in near-real-time (under 33 milliseconds for the engine to run at 30 frames per second), both AestheticCam and the deferred rendering game engine are unable to run in conjunction with one another without resulting in a decreased framerate. As such, the bulk of the camera optimization is run in a background thread via the queued threadpool architecture of Section 4.1.5.

AestheticCam is implemented as a Component that can be added to any CameraEntity defined by the game engine. This means that AestehticCam has access to the overall scene layout including the parent camera Entity. In order to ensure thread safety when accessing the scene or updating the camera attributes, the result of the background-threaded computation is dispatched to the main rendering thread. The next frame's computation kicked off again on a background thread, a process which continuously repeats as is depicted in Figure 4.4.

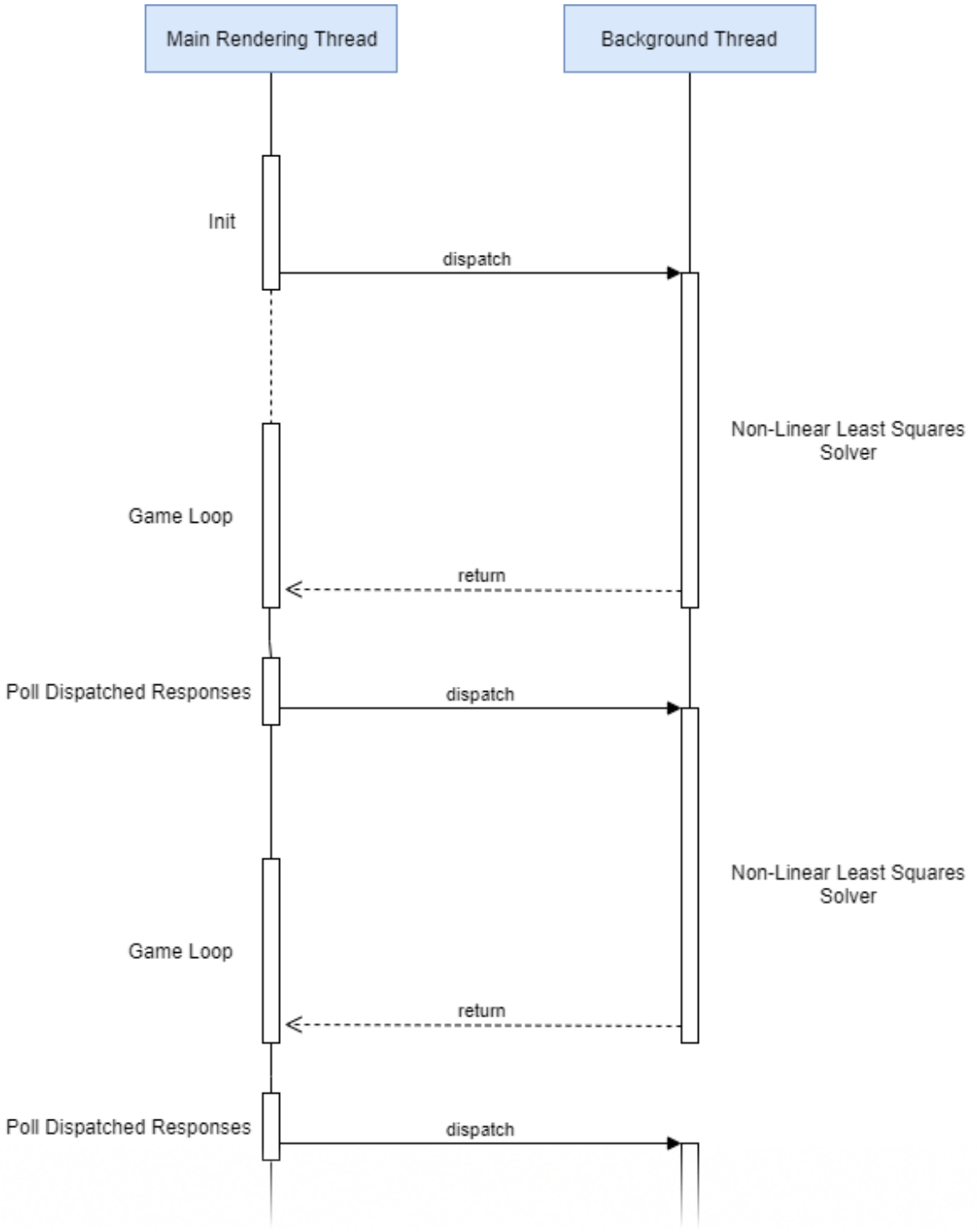


Figure 4.4: The basic relationship between the game engine and AestheticCam. All AestheticCam optimizations are computed on background threads. From there the results are dispatched to the main thread for rendering and the process repeats itself indefinitely.

4.3 Cost Functions

Encoding AestheticCam's behaviors as cost functions is at the heart of how effective it will be. Since cinematography is so subjective, the rule of thirds (see Section 2.5) was chosen as something that could easily be encoded as an objective cost function. In addition to the rule of thirds, game-specific logic for keeping the camera close to the player-controlled character was added as an additional cost function.

4.3.1 Rule of Thirds Cost

The game implemented for this thesis involves a player character pitted against a team of enemy characters. To adhere to the rule of thirds, the player character was placed in one of the bottom one-third marks and the team of enemy characters was placed in the diagonally-opposite one-third mark. A single cost function was constructed to optimally place an entity at a specific point within the camera frame.

Because rotation about Z is left fixed at 0 (see Section 4.2.1), a simplified version of Equation 2.1 is used for AestheticCam.

$$M = \begin{bmatrix} cos\theta & sin\theta sin\psi & sin\theta cos\psi & X \\ 0 & cos\psi & -sin\psi & Y \\ -sin\theta & cost\theta sin\psi & cos\theta cos\psi & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.7}$$

In essence, the cost function depicted in Algorithm 4.3 takes a point in three-dimensional space and projects it into the camera. We give the cost function a point that we want the projected point to minimize towards, so by subtracting the target position from the actual point, we get a function with a definite minimum residual value — that minimum at the target position. Knowing how Ceres' gradient descent

Algorithm 4.3: Cost function to place a three-dimensional point within the frame at a specific two-dimensional point

Parameters: X: Camera x-coordinate position.

Y: Camera y-coordinate position.

Z: Camera z-coordinate position.

ψ: Pitch (i.e. rotation about x-axis) of the camera.

θ: Yaw (i.e. rotation about y-axis) of the camera.

Static Inputs: P: Projection matrix for the current camera parameters (see Equation 2.3).

Target: Projected target location within normalized clip space from (-1, -1) to (1, 1).

Pt_{world}: Three-dimensional object point in world space to be projected into clip space.

Residuals: The final cost output of the function.

$M_{camera} \leftarrow$ ComputeModelTransform(X, Y, Z, ψ, θ)

V $\leftarrow M_{camera}^{-1}$

$Pt_{clip} \leftarrow$ P * V * Pt_{world}

$Pt_{normalized} \leftarrow Pt_{clip}$.xy / Pt_{clip}.w

Residuals $\leftarrow Pt_{normalized}$ - Target

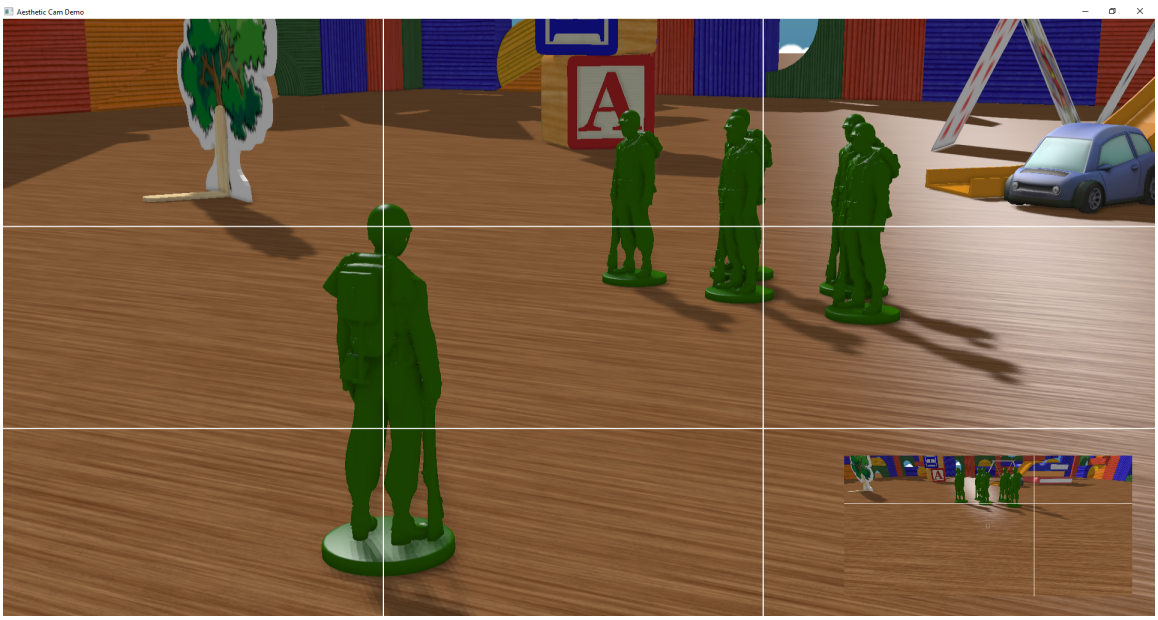


Figure 4.5: Here we see the result of the rule of thirds placement using the camera-projection cost functions. The player character is placed in the bottom-left portion of the frame with the entirety of the enemy team being loosely arranged around the top-right.

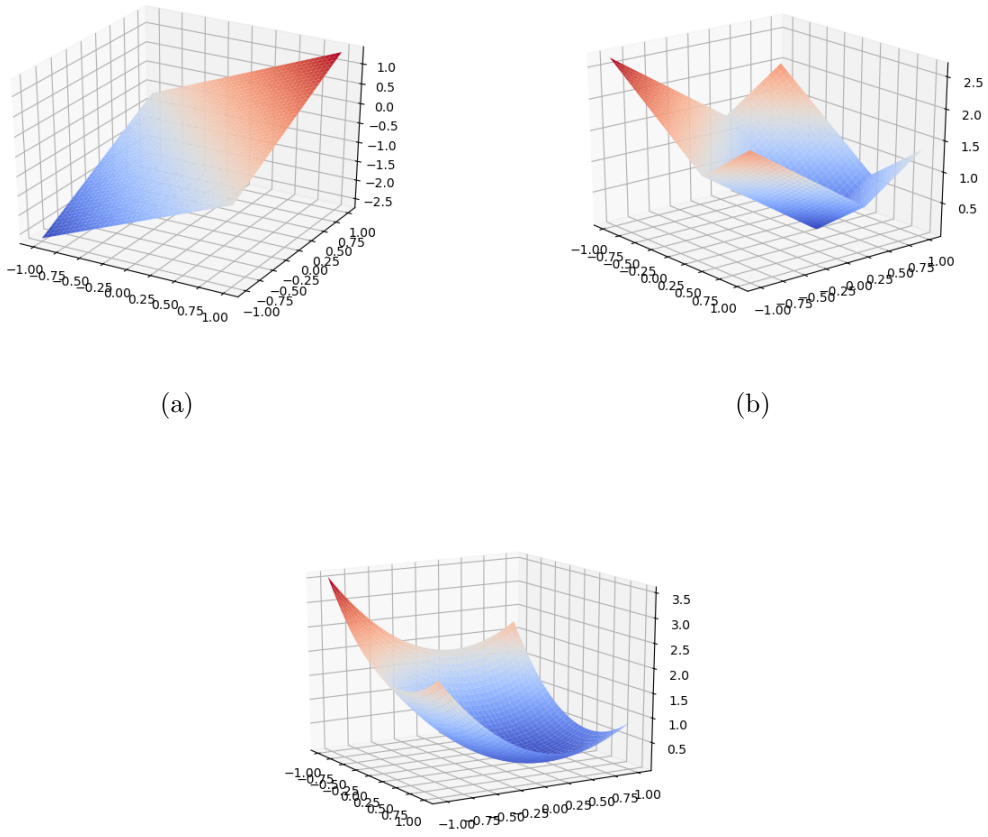
methods operate, we can intelligently create cost functions; for a concrete example, see Figure 4.6.

To place the player character in the bottom-left, a target position of (-33., -.33) was given to the cost function. Similarly, a cost function for each enemy characters was created and given a target position of (.33, 33) which equates to the top-right rule of thirds intersection.

4.3.2 Positional Cost

In addition to placing the player and enemy characters at rule of thirds locations, we also provide a cost function to keep the camera at a fixed distance from the player character.

To keep the camera "attached" to the player at a distance, we passed the player position as the target, and a fixed distance of 7 into the Algorithm 4.4 cost function.



(a) (b)

(c)

Figure 4.6: At face value, our cost function (a) appears to construct a plane without any discernible minimum point. A naive approach would be to create a cost function that returns an absolute value (b), but this approach creates discontinuities in the cost and actually makes it difficult for gradient descent to operate. If we remember that Ceres is a non-linear least squares optimizer, squares being the key operator, we see that the residuals returned from the cost function are squared (c) and in fact create a distinct minimum point.

Algorithm 4.4: Cost function to keep camera within a fixed distance of another point

Parameters: X: Camera x-coordinate position.

Y: Camera y-coordinate position.

Z: Camera z-coordinate position.

Static Inputs: Target: Three-dimensional target location in world space.

Distance: The distance the camera should be away from the target.

Residual: The final cost output of the function.

Delta $\leftarrow$ (X, Y, Z) - Target

Residual $\leftarrow$ Delta.LengthSquared() - Distance^2

At times, the rule of thirds cost function and distance cost function are at odds, sometimes causing one of the two to take priority. An optimized systems such as ours allows for this kind of flexibility, where rules can be broken and compromises can be met between mathematical constraints.

4.4 Summary

Our cinematic camera system, AestheticCam, utilizes non-linear least squares optimization to construct cinematic views. Similar to the algebraic system described in Section 3.1.1, we are able to codify a cinematic idiom - the rule of thirds - as a cost function that can be optimized. Using similar techniques to the constraint-based systems of Section 3.1.2, dynamic camera pathing and decision-making is achieved through the use of the Ceres Solver. Thanks to modern hardware and graphics standards, the result is a pleasing visual that can run in real-time with the player.

Figure 4.7: The distance cost function keeps the camera at a fixed distance from the player character, sometimes sacrificing rule of thirds adherence. This produces an image that is still comprehensible when the player character is close to the enemy characters.

Chapter 5

RESULTS

In order to fully validate AestheticCam from both a functional and aesthetic perspective, we developed a game engine and created a rudimentary first person game as described in fuller detail in Section 4.1.6. AestheticCam uses to paradigm of the team-based dynamic to construct its cinematic camera view.

5.1 Performance

A benchmark for usability of the AestheticCam system is that it is able to operate within modern performance standards. The faster it is able to optimize a cinematic view, the smoother the experience is for the viewer. To measure this, we ran timing tests on AestheticCam with commercial-grade hardware.

5.1.1 Hardware

The hardware used to validate the performance of the entire AestheticCam and rendering engine is as follows:

- CPU: AMD Ryzen 7, 8-Core multithreaded, 3.4GHz
- RAM: 32GB DDR4, 3200MHz
- GPU: Radeon R9 390, 8GB

5.1.2 Timing

In order validate how efficient AestheticCam operates, we timed how quickly the non-linear least squares optimization took on average over 100 iterations. We computed

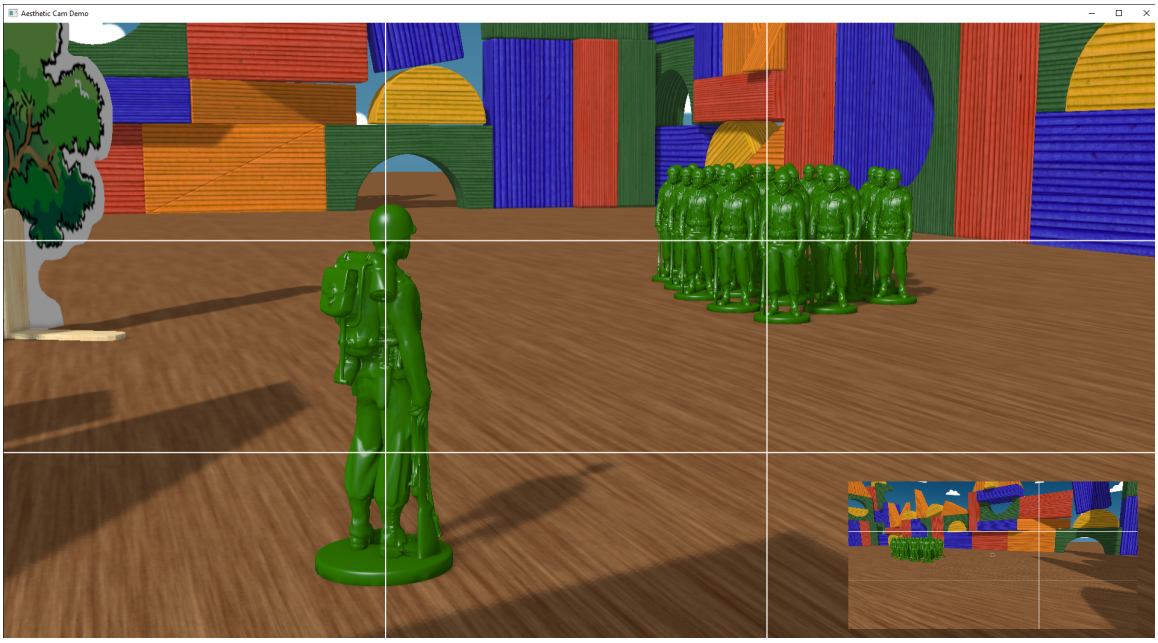


Figure 5.1: AestheticCam continues to produce valid rule of thirds results with upwards of 25+ enemy units, as seen here.

this average with an increasing number of enemy units, which also increased the number of cost functions that were added to the optimization problem (see Figure 5.1).

We see that as increasing the number of enemy units, Table 5.1, the time to compute decreases. The addition of further cost functions causes the optimizer to make increasingly more computations, which should increase the time to solve. However, due to enemy unit behavior, they tend to cluster together while following the player character. This plausibly attributes to the decreased computation time as the local minimum of the combined rule of thirds cost functions becomes more defined. However, this is merely speculation; finding the root cause of this behavior requires further inquiry into how Ceres optimizes its computations.

Table 5.1: Timing results of enemy count on AestheticCam performance.

Number of Enemies	Time to Optimize (ms)
1	91.9081
2	37.8011
3	25.4066
4	20.795
5	18.062
6	12.7348
7	11.2215
8	9.6009
9	10.7653
10	9.46106
20	5.36432
30	4.12445
40	3.1398
50	3.86062

5.2 Validation

Outside of technical performance, external validation was sought to determine if AestheticCam's results produced the desired effect of improved cinematic qualities. This validation took form as a set of two surveys, hereafter called Survey A and Survey B. Each survey provided four different viewpoints of the same piece of gameplay. These viewpoints were the player's first-person perspective, a third-person perspective locked to the player's rotation, a direct overhead view, and lastly the viewpoint from AestheticCam (see Figure 5.2). The two surveys, however, provided slightly different rule-of-thirds views; Survey A placed the player character in the bottom-left corner of the screen and the enemy team in the upper-right corner whereas Survey B placed the player in the bottom-left and the enemy team in the upper-left corner (see Figure 5.3).

The two surveys were designed to be identical except for the AestheticCam views with the intent of showing the robustness of the system. Participants for the surveys were gathered by reaching out online through social media connections both on Facebook and Reddit and waiting for self-reported responses. Overall, thirty-one responses were accumulated with eighteen of those for Survey A and fourteen for Survey B. The participants were also asked of their video game consumption habits, the results of which can be seen in Figure 5.4. As we can see, a vast majority of the participants were frequent players and viewers of video games. This can be largely attributed to the methods for which the participants were gathered, pooling from Cal Poly's computer science Facebook page as well as the SampleSize subreddit.

(a) First Person

(b) Third Person

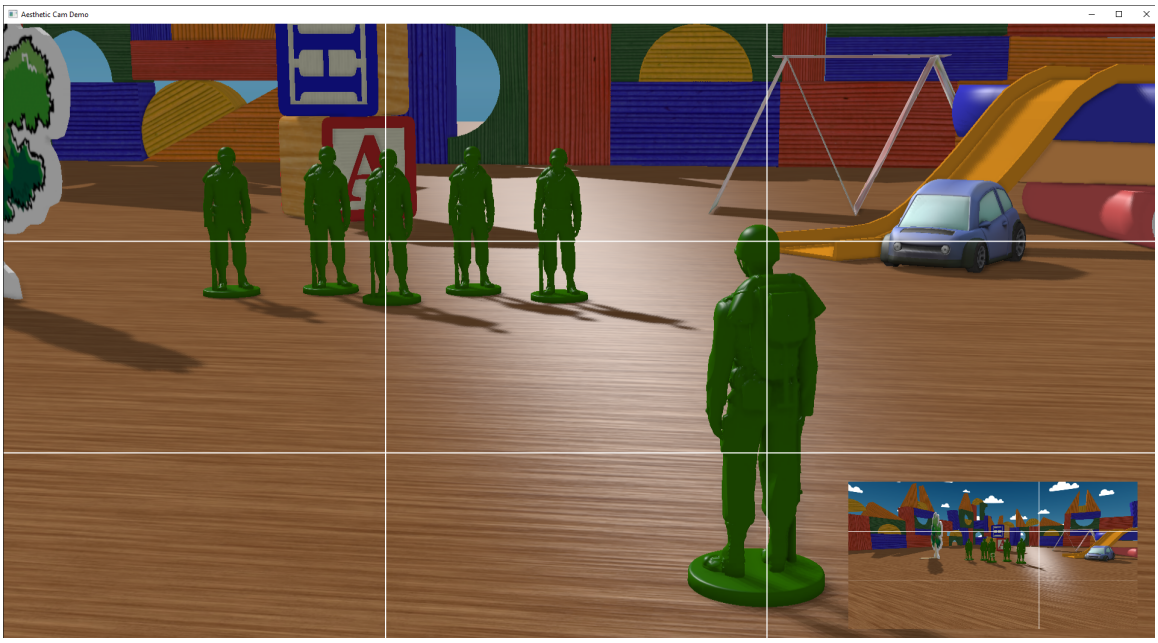
Figure 5.2: Comparison of the differing gameplay perspectives provided in the validation survey.

(c) Overhead

(d) AestheticCam

Figure 5.2: Comparison of the differing gameplay perspectives provided in the validation survey.

(a) Survey A



(b) Survey B

Figure 5.3: Comparison of the views AestheticCam provided in Survey A and Survey B.

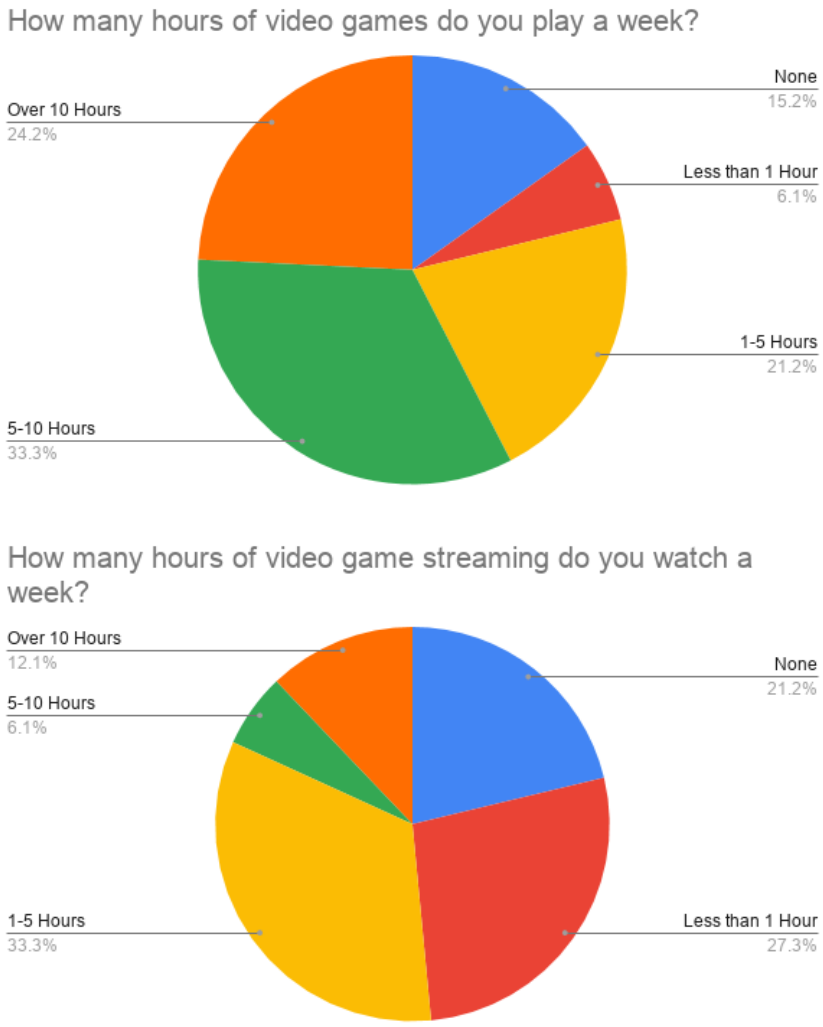


Figure 5.4: Breakdown of how much the participants consumed video games in a given week.

Table 5.2: Average points of each of the viewpoints after ranking point conversion.

	First Person	Cinematic	Third Person	Overhead
Question 1	3.129032258	2.580645161	3.032258065	1.258064516
Question 2	2.903225806	3.032258065	2.967741935	1.096774194
Question 3	3	2.741935484	3.096774194	1.161290323

5.2.1 Comparison Results

The participants of the surveys were instructed to watch the four different gameplay videos and evaluate them on an individual basis. Then at the end of the surveys, the four videos were asked to be directly compared and ranked against one another. The results from the two surveys can be see in Figure 5.5.

Quite clearly we can see that the overhead view was the general loser out of the four. This wasn't much of a surprise given that the perspective was both unconventional for the genre of game being played - a first-person shooter - as well as being unconventional for video games in general; most overhead views have a slight angle so as to include more in the frame.

The results of comparing the remaining three views is not so obvious; in many categories they appear to be equal. To perform more in-depth statistical analysis, the rankings were converted to a numerical value. Similar to ranked positional voting systems, a first preference was converted to 4 points, second preference was converted to 3 points and so on. The average point conversion can be seen in Table 5.2.

Looking at the averages, we can see that for Question 1 - ranking the views based on comprehension - the cinematic view loses out to both the First Person and Third Person viewpoints. In Question 2 - ranking the views based on aesthetic appeal -

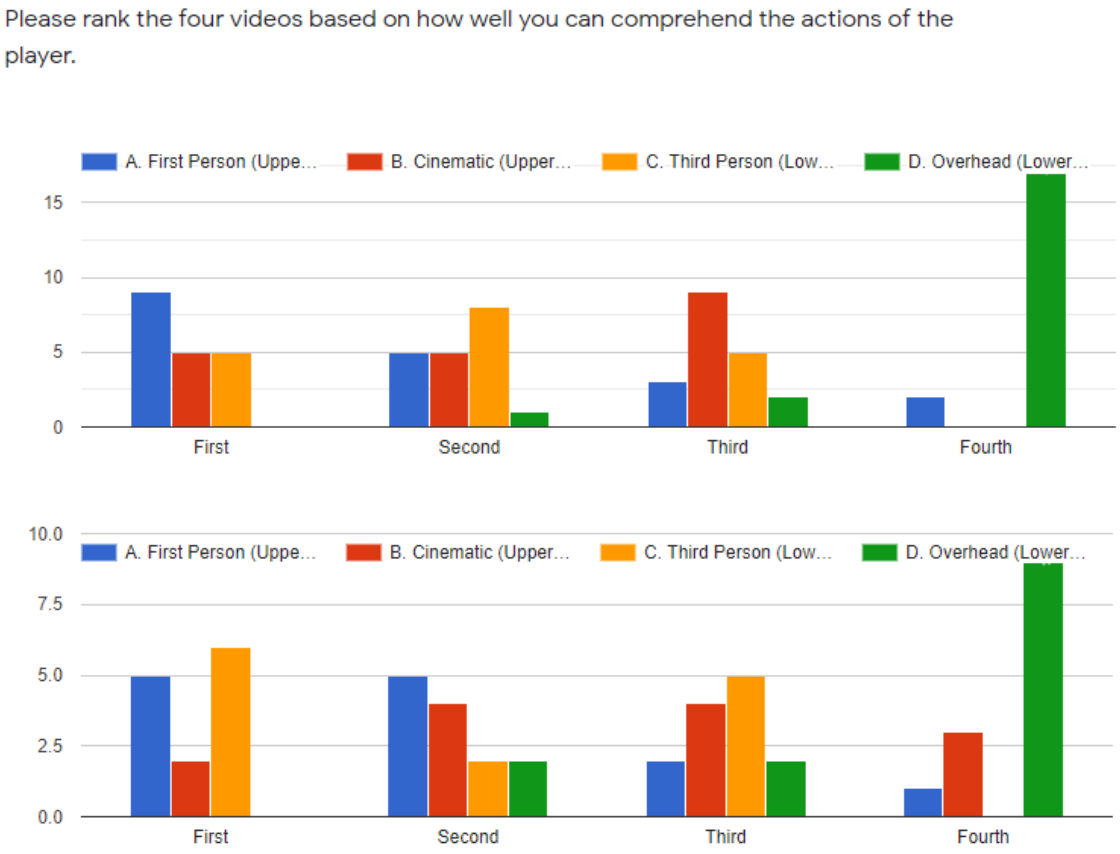


Figure 5.5: Question 1 results from being asked to directly rank the different views based on comprehension.

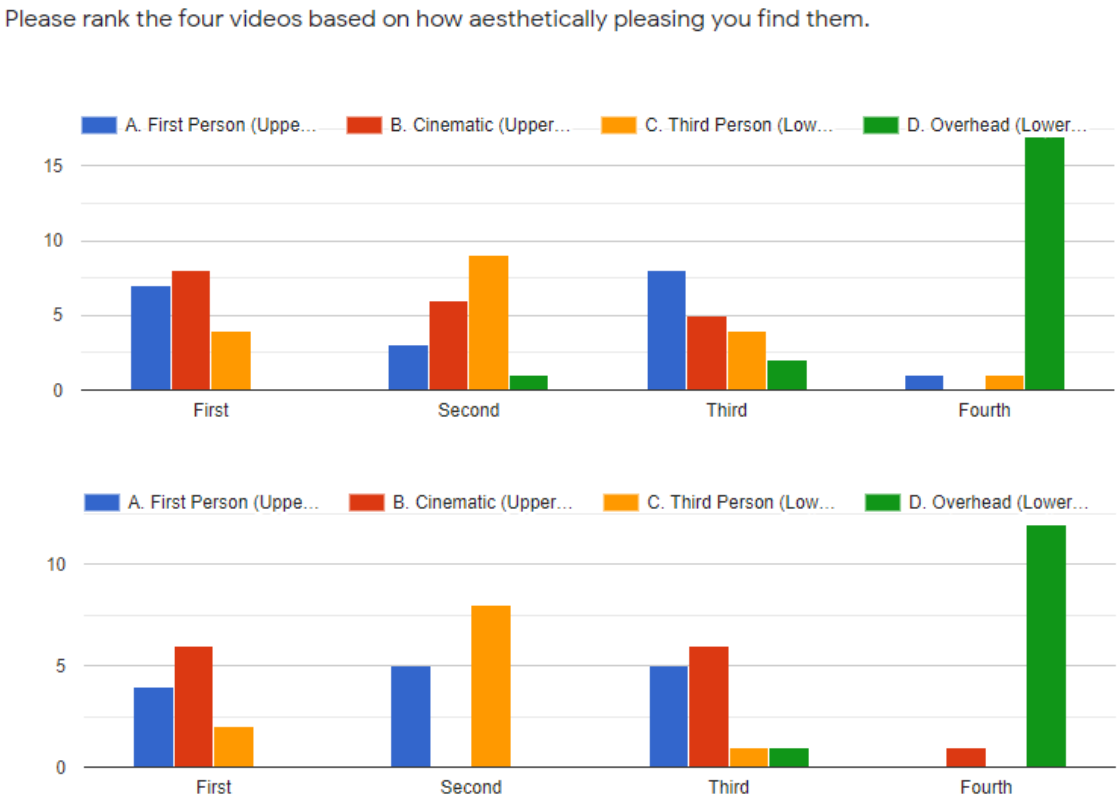


Figure 5.5: Question 2 results from being asked to directly rank the different views based on aesthetics.

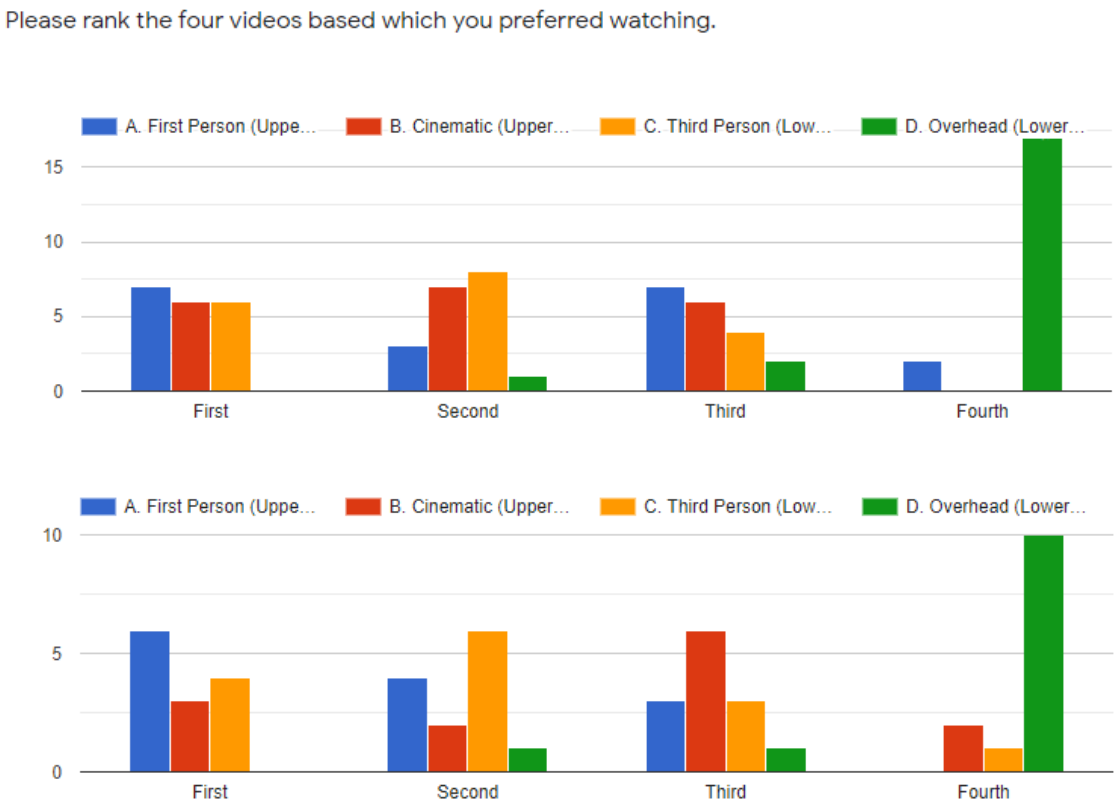


Figure 5.5: Question 3 results from being asked to directly rank the different views based on overall preference.

Table 5.3: Results of performing t-tests on the First Person and Third Person rankings against the Cinematic view generated from AestheticCam.

	First Person	Third Person
Question 1	0.06742387602	0.1045038087
Question 2	0.6837013837	0.7976867647
Question 3	0.4182446941	0.1405354933

the results of AestheticCam barely edge out the other two. Finally for Question 3, AestheticCam loses out to the other two views again.

However if we take our sample size and statistical variance into account, we see a slightly different story. To further compare each of our data sets, we performed a two-tailed t-test between the point-converted ranking values for the AestheticCam results and the First Person and Third Person data sets. By choosing the fairly standard (and generous) alpha value of $\alpha = .05$, we see that for every question and both of the other two views, we fail to establish any statistically significant difference between the average converted rankings of the First Person and Third Person view points versus the AestheticCam view point.

5.2.2 Analysis

Mostly due to our fairly sparse sample size of $n = 31$, we were unable to find any reasonably statistical significance difference between the rankings of the three view points. On one hand, AestheticCam was unable to establish itself as a superior strategy for generating spectator views for video games. Yet on the other hand, this partially implies that the results from AestheticCam were at least comparable on some level to the more standard and familiar views generated from the First Person and Third Person perspectives. Of course our approach to converting ranked preference to a number could partially be our downfall. Ranking does not imply linearity between

the rankings; the difference between an individual ranking first and second place could be much larger than between second and third for example. However we still believe this to be a valid approach to comparing rankings, especially with larger sample sizes.

We also aggregated our Survey A and Survey B results into a single ranking. When performing similar t-tests between the results of the two surveys, we found no indication that the rankings of the two surveys produced statically significant differences.

Chapter 6

CONCLUSION

By encoding the rule of thirds as a cost function in a non-linear least squares optimization library, we were able to design an automated cinematic camera system that can produce views that follow this rule. Although a fairly simple and objective metric, we were able to demonstrate that such a metric can at least stand on equal footing with more standard camera perspectives. Furthermore the groundwork here acts as a broad launchpad for other systems to utilize robust non-linear least squares optimizers and achieve more complex camera views. These systems can avoid the single-use of game or application-specific decision-making and work towards providing more general solutions to automating cinematography in games and real-time graphics applications.

6.1 Future Work

The visual results we see from AestheticCam as well as its promising performance on consumer hardware leave us with a long wish list of potential applications of this technology.

6.1.1 Game Types

The first-person shooter game that was developed for this thesis was very rudimentary, to put it nicely. However it *was* able to depict how AestheticCam would be applied to a first-person shooter in order to produce rule-of-thirds views. The full gamut of game genres and types is much wider than first-person shooters. An easy application

for AestheticCam would be to apply it to different game types, especially those that normally more closely fit the "Overhead" perspective generated for the surveys.

6.1.2 External Applications

Besides different game types, another proving point for AestheticCam would be to integrate with a production-ready game. Having to interface with a full-fledged engine that has all the bells and whistles of a triple-A title would require simplifying the API and would make sure the performance we see in our rendering engine is consistent across multiple contexts.

6.1.2.1 Unreal Engine

The easiest way to test AestheticCam on an external game would be to use a modern game engine and integrate it with some open source or sample game. Given Unreal Engine 4's open source and indie developer-friendly attitude, it is a prime candidate for next steps in terms of applying AestheticCam to something more complex. Additions to Unreal Engine can be directly written in C++, making porting over even that much simpler.

6.1.3 Additional Cinematic Metrics

Besides the rule of thirds, a lot of other cinematic angles are employed by cinematographers. Some of these metrics are to prevent unsightly frame compositions, such as preventing the edge of subjects from lining up with the very edge of the fame. There are also stylistic decisions that can be made, such as the more flat-symmetric aesthetic telltale of a Wes Anderson film. Some of these decisions can rely on the existing cost functions that AestheticCam already employs, like in-frame positioning

of objects, however many other cinematic shots would require further work encoding them as Ceres cost funcitons.

6.1.4 Additional Validation

Outside of development work, performing a much larger survey is certainly in order. A significant downside of the current survey is that we were unable to reasonably look at different demographic slices of the results. For example, looking at viewer preference for non-video game playing participants vs. more experienced participants was desired but the data just was not there. Such a comparison would tell us if taking a more cinematic approach to video game spectating could act as a lower barrier to entry for spectating games.

BIBLIOGRAPHY

[1] S. Agarwal, K. Mierle, and Others. Ceres solver. `http://ceres-solver.org`.

[2] W. H. Bares, J. P. Grégoire, and J. C. Lester. Realtime constraint-based cinematography for complex interactive 3d worlds. In *AAAI/IAAI*, pages 1101–1106, 1998.

[3] J. Blinn. Where am i? what am i looking at?(cinematography). *IEEE Computer Graphics and Applications*, 8(4):76–81, 1988.

[4] J. F. Blinn. Models of light reflection for computer synthesized pictures. In *ACM SIGGRAPH computer graphics*, volume 11, pages 192–198. ACM, 1977.

[5] B. Brown. *Cinematography: theory and practice: image making for cinematographers and directors*. Focal Press, 2016.

[6] D. B. Christianson, S. E. Anderson, L.-w. He, D. H. Salesin, D. S. Weld, and M. F. Cohen. Declarative camera control for automatic cinematography. In *AAAI/IAAI, Vol. 1*, pages 148–155, 1996.

[7] M. Christie, R. Machap, J.-M. Normand, P. Olivier, and J. Pickering. Virtual camera planning: A survey. In *International Symposium on Smart Graphics*, pages 40–52. Springer, 2005.

[8] R. Datta, D. Joshi, J. Li, and J. Z. Wang. Studying aesthetics in photographic images using a computational approach. In *European conference on computer vision*, pages 288–301. Springer, 2006.

[9] K. Davis. Probabilistic roadmaps for virtual camera pathing with cinematographic principles. 2017.

[10] S. M. Drucker, T. A. Galyean, and D. Zeltzer. Cinema: A system for procedural camera movements. In *Proceedings of the 1992 symposium on Interactive 3D graphics*, pages 67–70. ACM, 1992.

[11] S. M. Drucker and D. Zeltzer. Intelligent camera control in a virtual environment. In *Graphics Interface*, pages 190–190. CANADIAN INFORMATION PROCESSING SOCIETY, 1994.

[12] S. M. Drucker and D. Zeltzer. Camdroid: A system for implementing intelligent camera control. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 139–144. ACM, 1995.

[13] D. K. Elson and M. Riedl. A lightweight intelligent virtual cinematography system for machinima production. 2007.

[14] C. D. Encyclopedia. View frustum. http://encyclopedia2.thefreedictionary.com/View+frustum.

[15] Q. Galvane, J. Fleureau, F.-L. Tariolle, and P. Guillotel. Automated cinematography with unmanned aerial vehicles. *arXiv preprint arXiv:1712.04353*, 2017.

[16] C. Gough. Global esports market revenue 2022, Feb 2019.

[17] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566–580, 1996.

[18] K. Madsen, H. B. Nielsen, and O. Tingleff. Methods for non-linear least squares problems. 1999.

[19] L. Mai, H. Le, Y. Niu, and F. Liu. Rule of thirds detection from photograph. In *2011 IEEE International Symposium on Multimedia*, pages 91–96. IEEE, 2011.

[20] J. V. Mascelli. *The five C's of cinematography*. Grafic Publications, 1965.

[21] L. Neumann, M. Sbert, B. Gooch, W. Purgathofer, et al. Defining computational aesthetics. *Computational aesthetics in graphics, visualization and imaging*, pages 13–18, 2005.

[22] J. Nocedal and S. Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[23] J. Rigau, M. Feixas, and M. Sbert. Informational aesthetics measures. *IEEE Computer Graphics and Applications*, 28(2):24–34, 2008.

[24] O. Wiki. Rendering pipeline overview — opengl wiki, 2019. [Online].

APPENDIX

Appendix A

SURVEY

The survey conducted to validate the results of AestheticCam had two separate versions with different videos. These videos were functionally similar, however the content was slightly different, mainly with the AestheticCam-generated view, called "Cinematic", solving for slightly different views. The survey on the next few pages lists the videos that were presented to Survey A. To see the videos that were shown to Survey B, see Table A.1.

Table A.1: Videos used in the different survey versions.

	Survey A	Survey B
Third Person	`http://youtube.com/watch?v=vWWJ4eI0J_I`	`https://www.youtube.com/watch?v=aaniISFJ09Q`
First Person	`http://youtube.com/watch?v=j9CFvKRcXzk`	`https://www.youtube.com/watch?v=EzLJBortgkY`
Overhead	`http://youtube.com/watch?v=2TNRRyMqdbw`	`https://www.youtube.com/watch?v=bc2sg46VUEk`
Cinematic	`http://youtube.com/watch?v=dxQBD-KXF6o`	`https://www.youtube.com/watch?v=ubhl3X8wSXM`
Side-by-side	`http://youtube.com/watch?v=rpglWGzvA5s`	`https://www.youtube.com/watch?v=DWirEN5OO2k`

Spectating Video Games

This survey is a part of a research study being conducted by graduate student Ian Meeder and Dr. Zoë Wood within the Department of Computer Science at California Polytechnic State University, San Luis Obispo. The purpose of the study is to evaluate the effects of computer-generated camera animation.

You are asked to take part in this study by completing the following questionnaire. The questionnaire consists of a series of videos followed by a series of questions asking your opinions about the video. Each video is around thirty seconds with the entire questionnaire taking around five to ten minutes to complete. All responses are anonymous.

The videos consist of a rudimentary first person shooter game where a player-controlled character is followed by a squad of enemy AI-controlled characters. The player character can throw balls at the enemies, which removes them from the game on contact. You will be asked to evaluate these videos on how this gameplay is presented to you, rather than the gameplay itself.

For any questions about this study, feel free to reach out to Ian Meeder at meeder.ian@gmail.com.

Demographics

1. **How many hours of video games do you play a week?**
   Mark only one oval.

   - ◯ None
   - ◯ Less than 1 Hour
   - ◯ 1-5 Hours
   - ◯ 5-10 Hours
   - ◯ Over 10 Hours

2. **What are you favorite kinds of video games to play?**
   Check all that apply.

   - ☐ 2D/3D Platformers
   - ☐ Action/Adventure
   - ☐ First/Third Person Shooters
   - ☐ Fighting
   - ☐ Puzzle
   - ☐ RPGs
   - ☐ Simulations/Tycoons
   - ☐ Strategy
   - ☐ Racing
   - ☐ Sports
   - ☐ Other: _____

3. **How many hours of video game streaming do you watch a week?**
Mark only one oval.

- ◯ None
- ◯ Less than 1 Hour
- ◯ 1-5 Hours
- ◯ 5-10 Hours
- ◯ Over 10 Hours

4. **What are you favorite kinds of video games to watch?**
Check all that apply.

- ☐ 2D/3D Platformers
- ☐ Action/Adventure
- ☐ First/Third Person Shooters
- ☐ Fighting
- ☐ Puzzle
- ☐ RPGs
- ☐ Simulations/Tycoons
- ☐ Strategy
- ☐ Racing
- ☐ Sports
- ☐ Other: _____

Third Person (1/5)

http://youtube.com/watch?v=vWWJ4eI0J_I

Based on the video above, please agree or disagree with the following statements:

5. **The movement of the camera gave me a clear grasp on what the player was doing.**
   Mark only one oval.

	1	2	3	4	5	
Disagree	◯	◯	◯	◯	◯	Agree

6. **The camera's movement left me disoriented or confused at times.**
   Mark only one oval.

	1	2	3	4	5	
Disagree	◯	◯	◯	◯	◯	Agree

7. **I found the camera movement aesthetically pleasing.**
Mark only one oval.

	1	2	3	4	5	
Disagree	◯	◯	◯	◯	◯	Agree

8. **Additional Comments**

First Person (2/5)

http://youtube.com/watch?v=j9CFvKRcXzk

Based on the video above, please agree or disagree with the following statements:

9. **The movement of the camera gave me a clear grasp on what the player was doing.**
Mark only one oval.

	1	2	3	4	5	
Disagree	◯	◯	◯	◯	◯	Agree

10. **The camera's movement left me disoriented or confused at times.**
Mark only one oval.

	1	2	3	4	5	
Disagree	◯	◯	◯	◯	◯	Agree

11. **I found the camera movement aesthetically pleasing.**
Mark only one oval.

	1	2	3	4	5	
Disagree	◯	◯	◯	◯	◯	Agree

12. **Additional Comments**

Overhead (3/5)

http://youtube.com/watch?v=2TNRRyMqdbw

Based on the video above, please agree or disagree with the following statements:

13. **The movement of the camera gave me a clear grasp on what the player was doing.**
Mark only one oval.

	1	2	3	4	5	
Disagree	◯	◯	◯	◯	◯	Agree

14. **The camera's movement left me disoriented or confused at times.**
Mark only one oval.

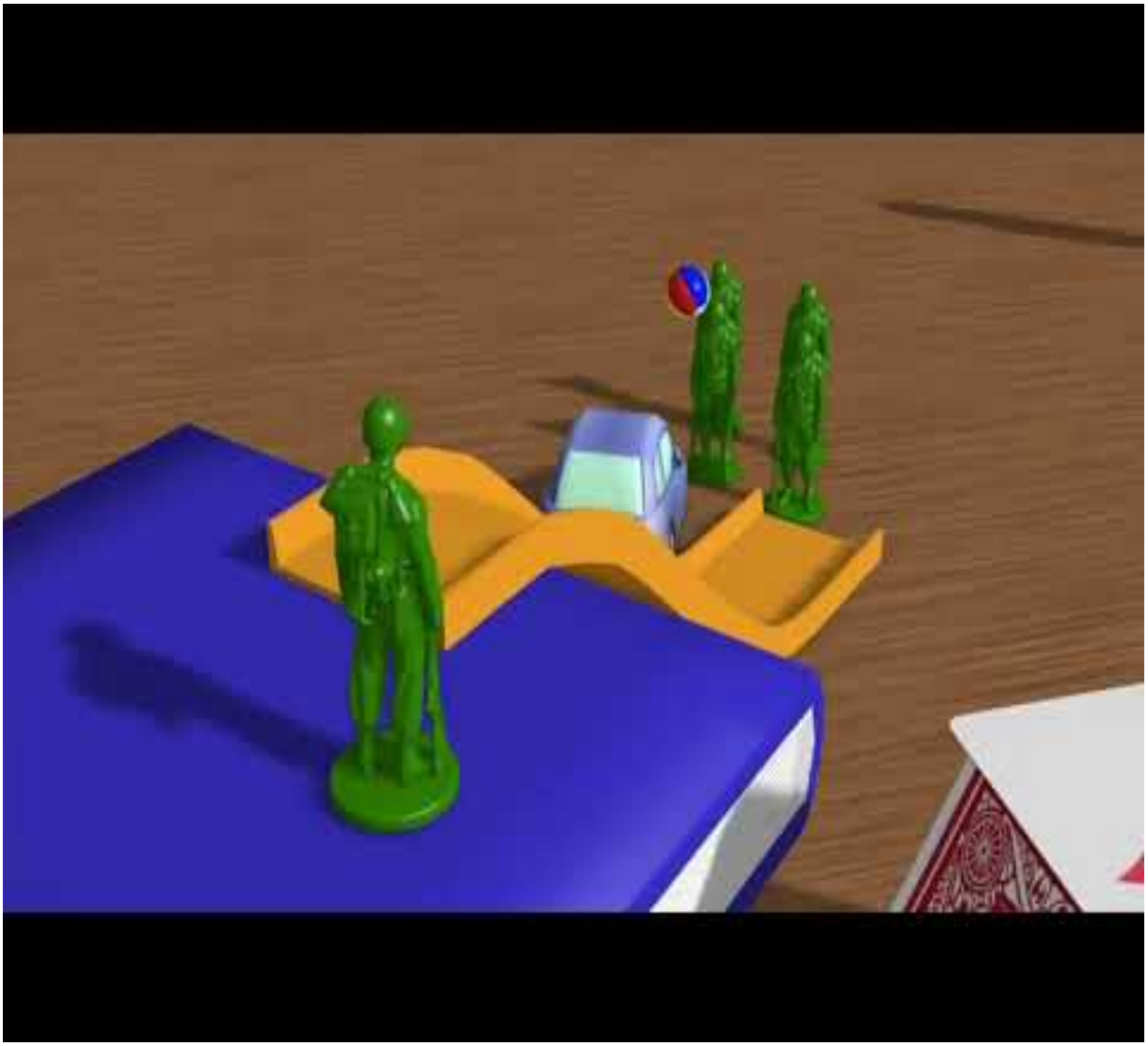
	1	2	3	4	5	
Disagree	◯	◯	◯	◯	◯	Agree

15. **I found the camera movement aesthetically pleasing.**
Mark only one oval.

	1	2	3	4	5	
Disagree	◯	◯	◯	◯	◯	Agree

16. **Additional Comments**

Cinematic (4/5)



http://youtube.com/watch?v=dxQBD-KXF6o

Based on the video above, please agree or disagree with the following statements:

17. **The movement of the camera gave me a clear grasp on what the player was doing.**
Mark only one oval.

	1	2	3	4	5	
Disagree	◯	◯	◯	◯	◯	Agree

18. **The camera's movement left me disoriented or confused at times.**
Mark only one oval.

	1	2	3	4	5	
Disagree	◯	◯	◯	◯	◯	Agree

19. **I found the camera movement aesthetically pleasing.**
Mark only one oval.

	1	2	3	4	5	
Disagree	◯	◯	◯	◯	◯	Agree

20. **Additional Comments**

Side-by-side (5/5)

http://youtube.com/watch?v=rpglWGzvA5s

You may have noticed that the four camera angles are of the exact same gameplay. Viewing them side by side, please rank them based on the following metrics:

21. **Please rank the four videos based on how well you can comprehend the actions of the player.**
Mark only one oval per row.

	A. First Person (Upper Left)	B. Cinematic (Upper Right)	C. Third Person (Lower Left)	D. Overhead (Lower Right)
First	◯	◯	◯	◯
Second	◯	◯	◯	◯
Third	◯	◯	◯	◯
Fourth	◯	◯	◯	◯

22. **Please rank the four videos based on how aesthetically pleasing you find them.**
Mark only one oval per row.

	A. First Person (Upper Left)	B. Cinematic (Upper Right)	C. Third Person (Lower Left)	D. Overhead (Lower Right)
First				
Second				
Third				
Fourth				

23. **Please rank the four videos based which you preferred watching.**
Mark only one oval per row.

	A. First Person (Upper Left)	B. Cinematic (Upper Right)	C. Third Person (Lower Left)	D. Overhead (Lower Right)
First				
Second				
Third				
Fourth				

24. **Additional Comments**